**DS Queue**

1. DS Queue
2. Types of Queues
3. Array Representation
4. Linked List Representation
5. Circular Queue
6. Deque
7. Priority Queue

# **Queue**

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



## Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

## Complexity

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Time Complexity** | | | | | | | | **Space Compleity** |
|  | **Average** | | | | **Worst** | | | | **Worst** |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| Queue | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

# **Types of Queue**

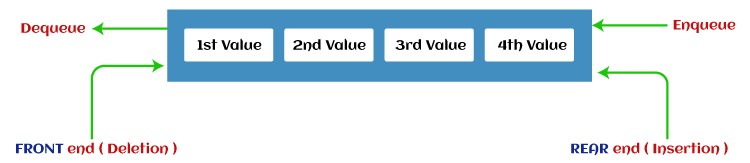
In this article, we will discuss the types of queue. But before moving towards the types, we should first discuss the brief introduction of the queue.

### **What is a Queue?**

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

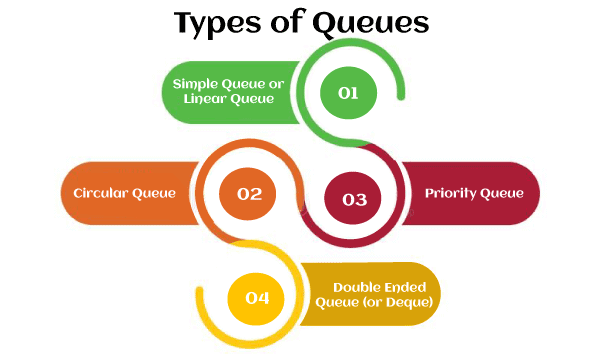
The representation of the queue is shown in the below image -



Now, let's move towards the types of queue.

### **Types of Queue**

There are four different types of queue that are listed as follows -

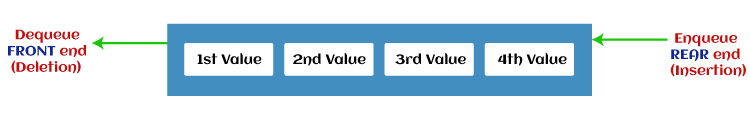


* Simple Queue or Linear Queue
* Circular Queue
* Priority Queue
* Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

### **Simple Queue or Linear Queue**

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

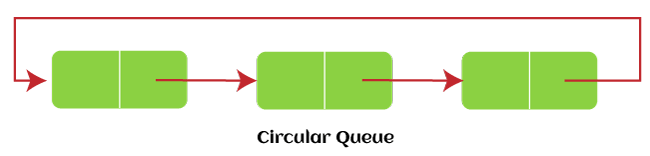


The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

To know more about the queue in data structure, you can click the link - <https://www.javatpoint.com/data-structure-queue>

### **Circular Queue**

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -

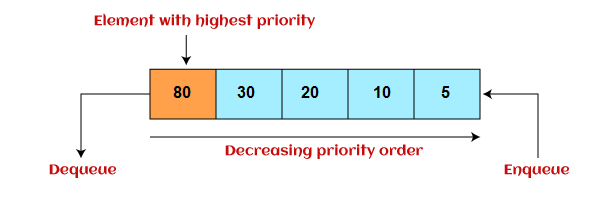


The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

To know more about the circular queue, you can click the link - <https://www.javatpoint.com/circular-queue>

### **Priority Queue**

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

* **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
* **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

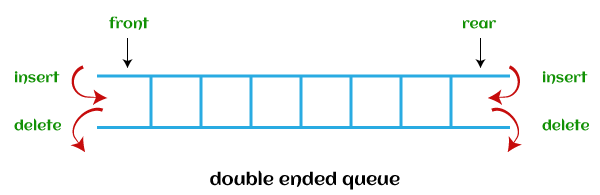
To learn more about the priority queue, you can click the link - <https://www.javatpoint.com/ds-priority-queue>

### **Deque (or, Double Ended Queue)**

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

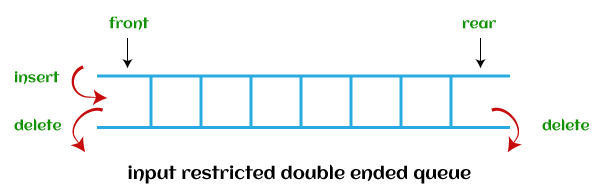
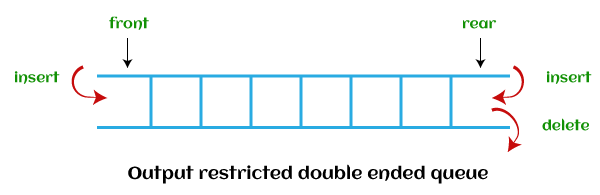
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -



To know more about the deque, you can click the link - <https://www.javatpoint.com/ds-deque>

There are two types of deque that are discussed as follows -

* **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.  
  
* **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.  
  

Now, let's see the operations performed on the queue.

## Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

* **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
* **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
* **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
* **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
* **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

## Ways to implement the queue

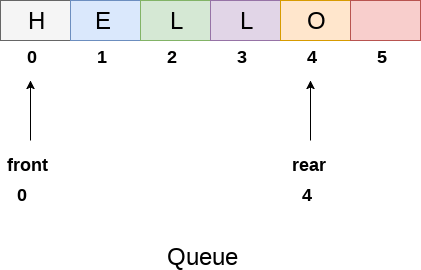
There are two ways of implementing the Queue:

* **Implementation using array:** The sequential allocation in a Queue can be implemented using an array. For more details, click on the below link: <https://www.javatpoint.com/array-representation-of-queue>
* **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked list. For more details, click on the below link: <https://www.javatpoint.com/linked-list-implementation-of-queue>

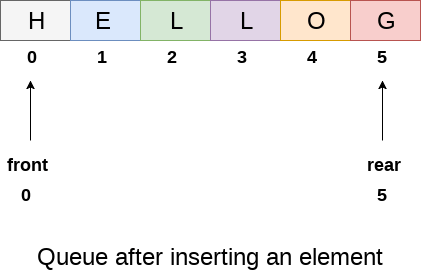
So, that's all about the article. Hope, the article will be helpful and informative to you.

# **Array representation of Queue**

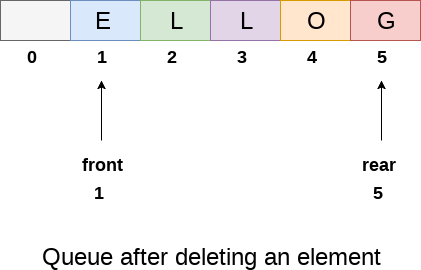
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



## Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

## Algorithm

* **Step 1:** IF REAR = MAX - 1  
  Write OVERFLOW  
  Go to step  
  [END OF IF]
* **Step 2:** IF FRONT = -1 and REAR = -1  
  SET FRONT = REAR = 0  
  ELSE  
  SET REAR = REAR + 1  
  [END OF IF]
* **Step 3:** Set QUEUE[REAR] = NUM
* **Step 4:** EXIT

## C Function

1. **void** insert (**int** queue[], **int** max, **int** front, **int** rear, **int** item)
2. {
3. **if** (rear + 1 == max)
4. {
5. printf("overflow");
6. }
7. **else**
8. {
9. **if**(front == -1 && rear == -1)
10. {
11. front = 0;
12. rear = 0;
13. }
14. **else**
15. {
16. rear = rear + 1;
17. }
18. queue[rear]=item;
19. }
20. }

## Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

## Algorithm

* **Step 1:** IF FRONT = -1 or FRONT > REAR  
  Write UNDERFLOW  
  ELSE  
  SET VAL = QUEUE[FRONT]  
  SET FRONT = FRONT + 1  
  [END OF IF]
* **Step 2:** EXIT

## C Function

1. **int** delete (**int** queue[], **int** max, **int** front, **int** rear)
2. {
3. **int** y;
4. **if** (front == -1 || front > rear)
6. {
7. printf("underflow");
8. }
9. **else**
10. {
11. y = queue[front];
12. **if**(front == rear)
13. {
14. front = rear = -1;
15. **else**
16. front = front + 1;
18. }
19. **return** y;
20. }
21. }

## Menu driven program to implement queue using array

1. #include<stdio.h>
2. #include<stdlib.h>
3. #define maxsize 5
4. **void** insert();
5. **void** delete();
6. **void** display();
7. **int** front = -1, rear = -1;
8. **int** queue[maxsize];
9. **void** main ()
10. {
11. **int** choice;
12. **while**(choice != 4)
13. {
14. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
15. printf("\n=================================================================\n");
16. printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
17. printf("\nEnter your choice ?");
18. scanf("%d",&choice);
19. **switch**(choice)
20. {
21. **case** 1:
22. insert();
23. **break**;
24. **case** 2:
25. delete();
26. **break**;
27. **case** 3:
28. display();
29. **break**;
30. **case** 4:
31. exit(0);
32. **break**;
33. **default**:
34. printf("\nEnter valid choice??\n");
35. }
36. }
37. }
38. **void** insert()
39. {
40. **int** item;
41. printf("\nEnter the element\n");
42. scanf("\n%d",&item);
43. **if**(rear == maxsize-1)
44. {
45. printf("\nOVERFLOW\n");
46. **return**;
47. }
48. **if**(front == -1 && rear == -1)
49. {
50. front = 0;
51. rear = 0;
52. }
53. **else**
54. {
55. rear = rear+1;
56. }
57. queue[rear] = item;
58. printf("\nValue inserted ");
60. }
61. **void** delete()
62. {
63. **int** item;
64. **if** (front == -1 || front > rear)
65. {
66. printf("\nUNDERFLOW\n");
67. **return**;
69. }
70. **else**
71. {
72. item = queue[front];
73. **if**(front == rear)
74. {
75. front = -1;
76. rear = -1 ;
77. }
78. **else**
79. {
80. front = front + 1;
81. }
82. printf("\nvalue deleted ");
83. }

86. }
88. **void** display()
89. {
90. **int** i;
91. **if**(rear == -1)
92. {
93. printf("\nEmpty queue\n");
94. }
95. **else**
96. {   printf("\nprinting values .....\n");
97. **for**(i=front;i<=rear;i++)
98. {
99. printf("\n%d\n",queue[i]);
100. }
101. }
102. }

**Output:**

\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*

==============================================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter the element

123

Value inserted

\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*

==============================================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter the element

90

Value inserted

\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*

===================================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?2

value deleted

\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*

==============================================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

90

\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*

==============================================

1.insert an element

2.Delete an element

3.Display the queue

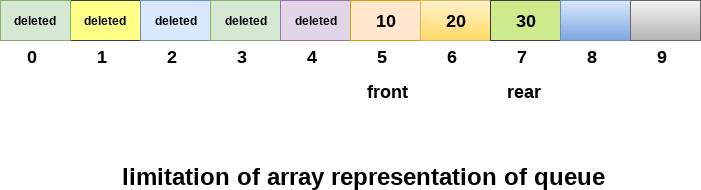
4.Exit

Enter your choice ?4

## Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

* **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

* **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

# **Linked List implementation of Queue**

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

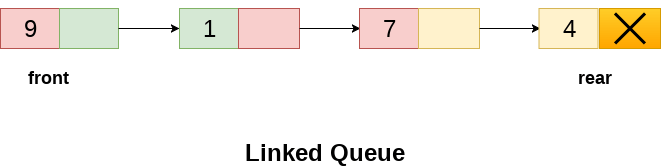
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



## Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

## Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node \*) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

1. ptr -> data = item;
2. **if**(front == NULL)
3. {
4. front = ptr;
5. rear = ptr;
6. front -> next = NULL;
7. rear -> next = NULL;
8. }

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

1. rear -> next = ptr;
2. rear = ptr;
3. rear->next = NULL;

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

## Algorithm

* **Step 1:** Allocate the space for the new node PTR
* **Step 2:** SET PTR -> DATA = VAL
* **Step 3:** IF FRONT = NULL  
  SET FRONT = REAR = PTR  
  SET FRONT -> NEXT = REAR -> NEXT = NULL  
  ELSE  
  SET REAR -> NEXT = PTR  
  SET REAR = PTR  
  SET REAR -> NEXT = NULL  
  [END OF IF]
* **Step 4:** END

## C Function

1. **void** insert(struct node \*ptr, **int** item; )
2. {

5. ptr = (struct node \*) malloc (sizeof(struct node));
6. **if**(ptr == NULL)
7. {
8. printf("\nOVERFLOW\n");
9. **return**;
10. }
11. **else**
12. {
13. ptr -> data = item;
14. **if**(front == NULL)
15. {
16. front = ptr;
17. rear = ptr;
18. front -> next = NULL;
19. rear -> next = NULL;
20. }
21. **else**
22. {
23. rear -> next = ptr;
24. rear = ptr;
25. rear->next = NULL;
26. }
27. }
28. }

## Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

1. ptr = front;
2. front = front -> next;
3. free(ptr);

The algorithm and C function is given as follows.

## Algorithm

* **Step 1:** IF FRONT = NULL  
  Write " Underflow "  
  Go to Step 5  
  [END OF IF]
* **Step 2:** SET PTR = FRONT
* **Step 3:** SET FRONT = FRONT -> NEXT
* **Step 4:** FREE PTR
* **Step 5:** END

## C Function

1. **void** delete (struct node \*ptr)
2. {
3. **if**(front == NULL)
4. {
5. printf("\nUNDERFLOW\n");
6. **return**;
7. }
8. **else**
9. {
10. ptr = front;
11. front = front -> next;
12. free(ptr);
13. }
14. }

## Menu-Driven Program implementing all the operations on Linked Queue

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5. **int** data;
6. struct node \*next;
7. };
8. struct node \*front;
9. struct node \*rear;
10. **void** insert();
11. **void** delete();
12. **void** display();
13. **void** main ()
14. {
15. **int** choice;
16. **while**(choice != 4)
17. {
18. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
19. printf("\n=================================================================\n");
20. printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
21. printf("\nEnter your choice ?");
22. scanf("%d",& choice);
23. **switch**(choice)
24. {
25. **case** 1:
26. insert();
27. **break**;
28. **case** 2:
29. delete();
30. **break**;
31. **case** 3:
32. display();
33. **break**;
34. **case** 4:
35. exit(0);
36. **break**;
37. **default**:
38. printf("\nEnter valid choice??\n");
39. }
40. }
41. }
42. **void** insert()
43. {
44. struct node \*ptr;
45. **int** item;
47. ptr = (struct node \*) malloc (sizeof(struct node));
48. **if**(ptr == NULL)
49. {
50. printf("\nOVERFLOW\n");
51. **return**;
52. }
53. **else**
54. {
55. printf("\nEnter value?\n");
56. scanf("%d",&item);
57. ptr -> data = item;
58. **if**(front == NULL)
59. {
60. front = ptr;
61. rear = ptr;
62. front -> next = NULL;
63. rear -> next = NULL;
64. }
65. **else**
66. {
67. rear -> next = ptr;
68. rear = ptr;
69. rear->next = NULL;
70. }
71. }
72. }
73. **void** delete ()
74. {
75. struct node \*ptr;
76. **if**(front == NULL)
77. {
78. printf("\nUNDERFLOW\n");
79. **return**;
80. }
81. **else**
82. {
83. ptr = front;
84. front = front -> next;
85. free(ptr);
86. }
87. }
88. **void** display()
89. {
90. struct node \*ptr;
91. ptr = front;
92. **if**(front == NULL)
93. {
94. printf("\nEmpty queue\n");
95. }
96. **else**
97. {   printf("\nprinting values .....\n");
98. **while**(ptr != NULL)
99. {
100. printf("\n%d\n",ptr -> data);
101. ptr = ptr -> next;
102. }
103. }
104. }

**Output:**

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

123

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

90

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

123

90

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?2

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

90

\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*

==============================

1.insert an element

2.Delete an element

3.Display the queue

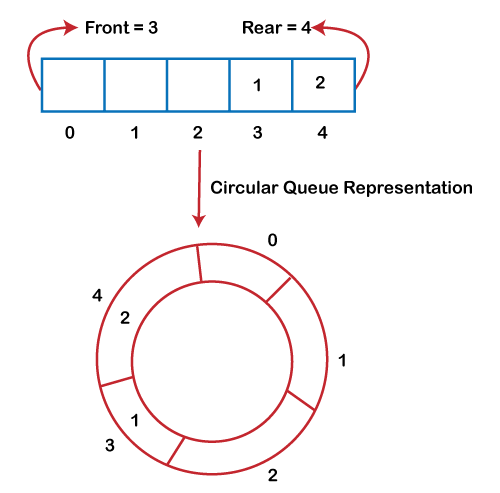
4.Exit

Enter your choice ?4

# **Circular Queue**

### **Why was the concept of the circular queue introduced?**

There was one limitation in the array implementation of [Queue](https://www.javatpoint.com/data-structure-queue). If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

### **What is a Circular Queue?**

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a **Ring Buffer**.

### **Operations on Circular Queue**

The following are the operations that can be performed on a circular queue:

* **Front:** It is used to get the front element from the Queue.
* **Rear:** It is used to get the rear element from the Queue.
* **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
* **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

### **Applications of Circular Queue**

**The circular Queue can be used in the following scenarios:**

* **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
* **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
* **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

### **Enqueue operation**

**The steps of enqueue operation are given below:**

* First, we will check whether the Queue is full or not.
* Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
* When we insert a new element, the rear gets incremented, i.e., **rear=rear+1**.

### **Scenarios for inserting an element**

**There are two scenarios in which queue is not full:**

* **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
* **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

**There are two cases in which the element cannot be inserted:**

* When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
* front== rear + 1;

**Algorithm to insert an element in a circular queue**

**Step 1:** IF (REAR+1)%MAX = FRONT  
Write " OVERFLOW "  
Goto step 4  
[End OF IF]

**Step 2:** IF FRONT = -1 and REAR = -1  
SET FRONT = REAR = 0  
ELSE IF REAR = MAX - 1 and FRONT ! = 0  
SET REAR = 0  
ELSE  
SET REAR = (REAR + 1) % MAX  
[END OF IF]

**Step 3:** SET QUEUE[REAR] = VAL

**Step 4:** EXIT

### **Dequeue Operation**

The steps of dequeue operation are given below:

* First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
* When the element is deleted, the value of front gets decremented by 1.
* If there is only one element left which is to be deleted, then the front and rear are reset to -1.

**Algorithm to delete an element from the circular queue**

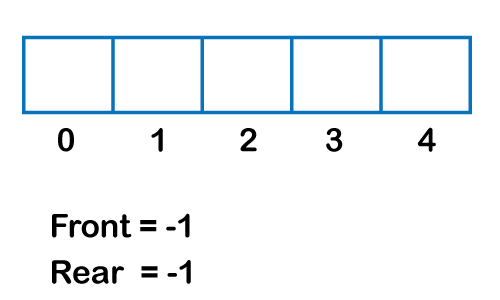
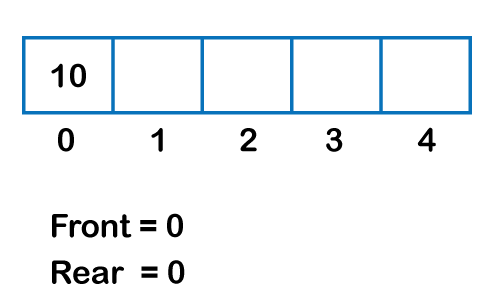
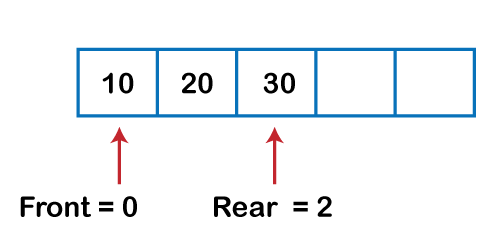
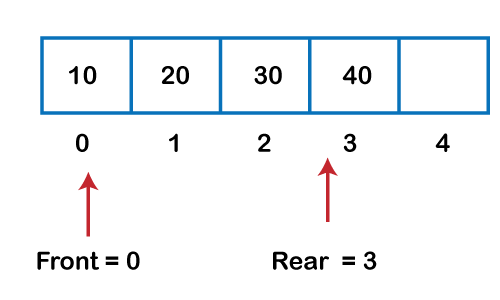
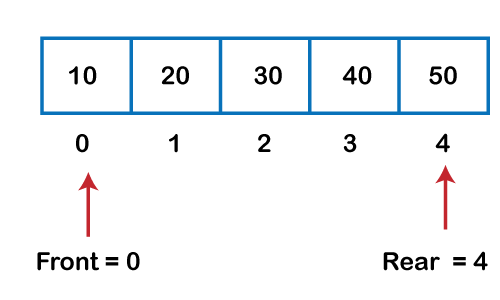
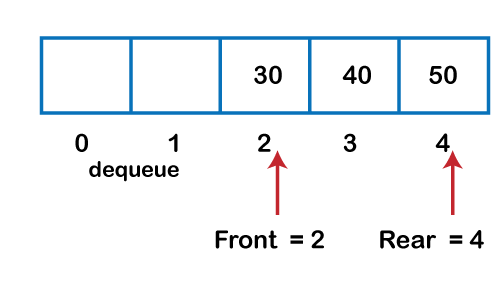
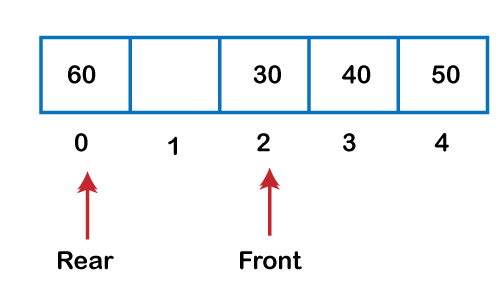
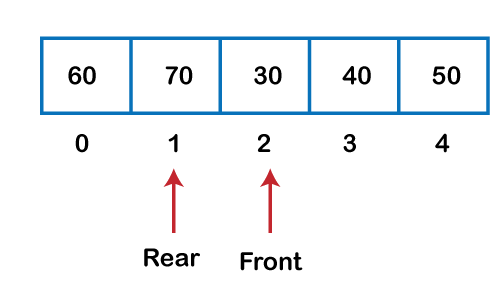
**Step 1:** IF FRONT = -1  
Write " UNDERFLOW "  
Goto Step 4  
[END of IF]

**Step 2:** SET VAL = QUEUE[FRONT]

**Step 3:** IF FRONT = REAR  
SET FRONT = REAR = -1  
ELSE  
IF FRONT = MAX -1  
SET FRONT = 0  
ELSE  
SET FRONT = FRONT + 1  
[END of IF]  
[END OF IF]

**Step 4:** EXIT

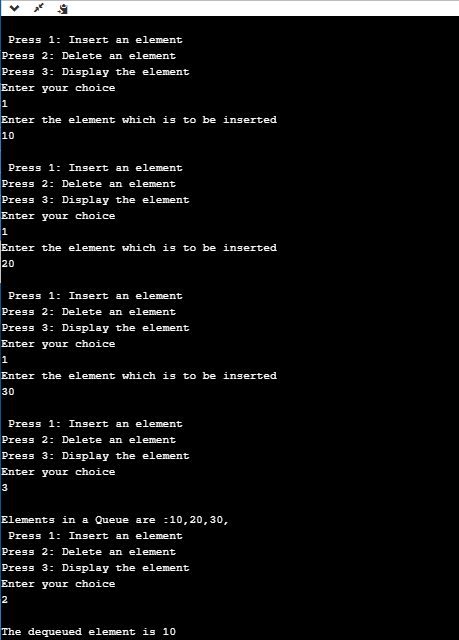
**Let's understand the enqueue and dequeue operation through the diagrammatic representation.**

### **Implementation of circular queue using Array**

1. #include <stdio.h>
3. # define max 6
4. **int** queue[max];  // array declaration
5. **int** front=-1;
6. **int** rear=-1;
7. // function to insert an element in a circular queue
8. **void** enqueue(**int** element)
9. {
10. **if**(front==-1 && rear==-1)   // condition to check queue is empty
11. {
12. front=0;
13. rear=0;
14. queue[rear]=element;
15. }
16. **else** **if**((rear+1)%max==front)  // condition to check queue is full
17. {
18. printf("Queue is overflow..");
19. }
20. **else**
21. {
22. rear=(rear+1)%max;       // rear is incremented
23. queue[rear]=element;     // assigning a value to the queue at the rear position.
24. }
25. }
27. // function to delete the element from the queue
28. **int** dequeue()
29. {
30. **if**((front==-1) && (rear==-1))  // condition to check queue is empty
31. {
32. printf("\nQueue is underflow..");
33. }
34. **else** **if**(front==rear)
35. {
36. printf("\nThe dequeued element is %d", queue[front]);
37. front=-1;
38. rear=-1;
39. }
40. **else**
41. {
42. printf("\nThe dequeued element is %d", queue[front]);
43. front=(front+1)%max;
44. }
45. }
46. // function to display the elements of a queue
47. **void** display()
48. {
49. **int** i=front;
50. **if**(front==-1 && rear==-1)
51. {
52. printf("\n Queue is empty..");
53. }
54. **else**
55. {
56. printf("\nElements in a Queue are :");
57. **while**(i<=rear)
58. {
59. printf("%d,", queue[i]);
60. i=(i+1)%max;
61. }
62. }
63. }
64. **int** main()
65. {
66. **int** choice=1,x;   // variables declaration
68. **while**(choice<4 && choice!=0)   // while loop
69. {
70. printf("\n Press 1: Insert an element");
71. printf("\nPress 2: Delete an element");
72. printf("\nPress 3: Display the element");
73. printf("\nEnter your choice");
74. scanf("%d", &choice);
76. **switch**(choice)
77. {
79. **case** 1:
81. printf("Enter the element which is to be inserted");
82. scanf("%d", &x);
83. enqueue(x);
84. **break**;
85. **case** 2:
86. dequeue();
87. **break**;
88. **case** 3:
89. display();
91. }}
92. **return** 0;
93. }

**Output:**

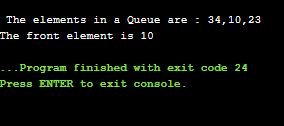


### **Implementation of circular queue using linked list**

As we know that linked list is a linear data structure that stores two parts, i.e., data part and the address part where address part contains the address of the next node. Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue. When we are implementing the circular queue using linked list then both the **enqueue and dequeue** operations take **O(1)** time.

1. #include <stdio.h>
2. // Declaration of struct type node
3. **struct** node
4. {
5. **int** data;
6. **struct** node \*next;
7. };
8. **struct** node \*front=-1;
9. **struct** node \*rear=-1;
10. // function to insert the element in the Queue
11. **void** enqueue(**int** x)
12. {
13. **struct** node \*newnode;  // declaration of pointer of struct node type.
14. newnode=(**struct** node \*)malloc(**sizeof**(**struct** node));  // allocating the memory to the newnode
15. newnode->data=x;
16. newnode->next=0;
17. **if**(rear==-1)  // checking whether the Queue is empty or not.
18. {
19. front=rear=newnode;
20. rear->next=front;
21. }
22. **else**
23. {
24. rear->next=newnode;
25. rear=newnode;
26. rear->next=front;
27. }
28. }
30. // function to delete the element from the queue
31. **void** dequeue()
32. {
33. **struct** node \*temp;   // declaration of pointer of node type
34. temp=front;
35. **if**((front==-1)&&(rear==-1))  // checking whether the queue is empty or not
36. {
37. printf("\nQueue is empty");
38. }
39. **else** **if**(front==rear)  // checking whether the single element is left in the queue
40. {
41. front=rear=-1;
42. free(temp);
43. }
44. **else**
45. {
46. front=front->next;
47. rear->next=front;
48. free(temp);
49. }
50. }
52. // function to get the front of the queue
53. **int** peek()
54. {
55. **if**((front==-1) &&(rear==-1))
56. {
57. printf("\nQueue is empty");
58. }
59. **else**
60. {
61. printf("\nThe front element is %d", front->data);
62. }
63. }
65. // function to display all the elements of the queue
66. **void** display()
67. {
68. **struct** node \*temp;
69. temp=front;
70. printf("\n The elements in a Queue are : ");
71. **if**((front==-1) && (rear==-1))
72. {
73. printf("Queue is empty");
74. }
76. **else**
77. {
78. **while**(temp->next!=front)
79. {
80. printf("%d,", temp->data);
81. temp=temp->next;
82. }
83. printf("%d", temp->data);
84. }
85. }
87. **void** main()
88. {
89. enqueue(34);
90. enqueue(10);
91. enqueue(23);
92. display();
93. dequeue();
94. peek();
95. }

**Output:**



# **Deque (or double-ended queue)**

In this article, we will discuss the double-ended queue or deque. We should first see a brief description of the queue.

### **What is a queue?**

A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail,** whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the person enters last in the queue gets the ticket at last.

### **What is a Deque (or double-ended queue)**

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -



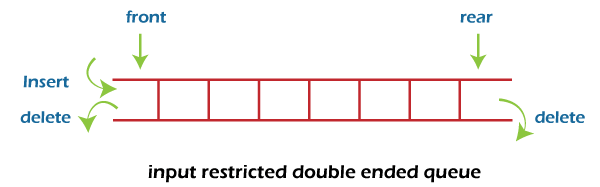
### **Types of deque**

There are two types of deque -

* Input restricted queue
* Output restricted queue

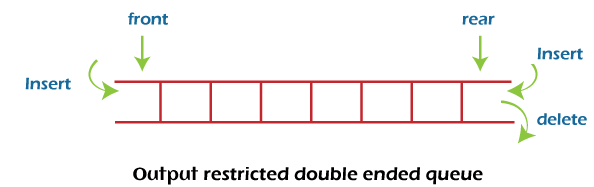
**Input restricted Queue**

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



**Output restricted Queue**

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



### **Operations performed on deque**

There are the following operations that can be applied on a deque -

* Insertion at front
* Insertion at rear
* Deletion at front
* Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

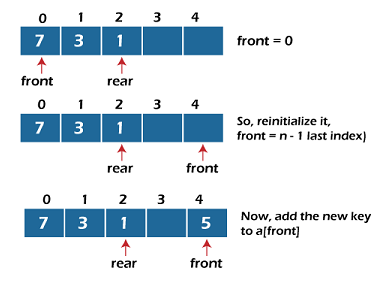
* Get the front item from the deque
* Get the rear item from the deque
* Check whether the deque is full or not
* Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

**Insertion at the front end**

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

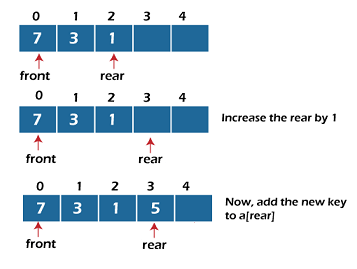
* If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
* Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



**Insertion at the rear end**

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

* If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
* Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



**Deletion at the front end**

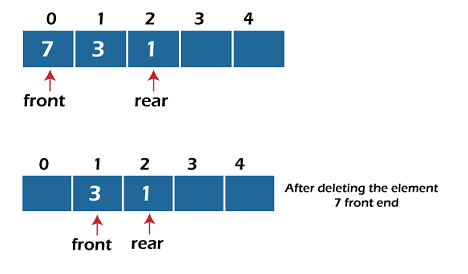
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



**Deletion at the rear end**

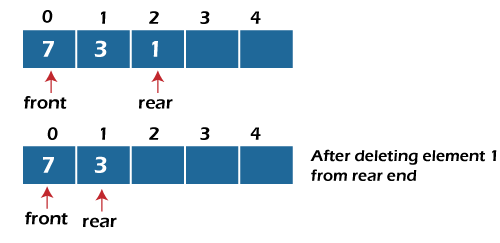
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



**Check empty**

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

**Check full**

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

## ****Applications of deque****

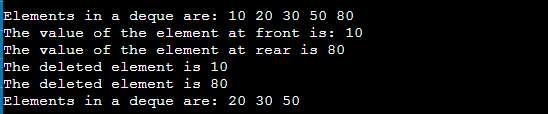
* Deque can be used as both stack and queue, as it supports both operations.
* Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

## Implementation of deque

Now, let's see the implementation of deque in C programming language.

1. #include <stdio.h>
2. #define size 5
3. **int** deque[size];
4. **int** f = -1, r = -1;
5. //  insert\_front function will insert the value from the front
6. **void** insert\_front(**int** x)
7. {
8. **if**((f==0 && r==size-1) || (f==r+1))
9. {
10. printf("Overflow");
11. }
12. **else** **if**((f==-1) && (r==-1))
13. {
14. f=r=0;
15. deque[f]=x;
16. }
17. **else** **if**(f==0)
18. {
19. f=size-1;
20. deque[f]=x;
21. }
22. **else**
23. {
24. f=f-1;
25. deque[f]=x;
26. }
27. }
29. // insert\_rear function will insert the value from the rear
30. **void** insert\_rear(**int** x)
31. {
32. **if**((f==0 && r==size-1) || (f==r+1))
33. {
34. printf("Overflow");
35. }
36. **else** **if**((f==-1) && (r==-1))
37. {
38. r=0;
39. deque[r]=x;
40. }
41. **else** **if**(r==size-1)
42. {
43. r=0;
44. deque[r]=x;
45. }
46. **else**
47. {
48. r++;
49. deque[r]=x;
50. }
52. }
54. // display function prints all the value of deque.
55. **void** display()
56. {
57. **int** i=f;
58. printf("\nElements in a deque are: ");
60. **while**(i!=r)
61. {
62. printf("%d ",deque[i]);
63. i=(i+1)%size;
64. }
65. printf("%d",deque[r]);
66. }
68. // getfront function retrieves the first value of the deque.
69. **void** getfront()
70. {
71. **if**((f==-1) && (r==-1))
72. {
73. printf("Deque is empty");
74. }
75. **else**
76. {
77. printf("\nThe value of the element at front is: %d", deque[f]);
78. }
80. }
82. // getrear function retrieves the last value of the deque.
83. **void** getrear()
84. {
85. **if**((f==-1) && (r==-1))
86. {
87. printf("Deque is empty");
88. }
89. **else**
90. {
91. printf("\nThe value of the element at rear is %d", deque[r]);
92. }
94. }
96. // delete\_front() function deletes the element from the front
97. **void** delete\_front()
98. {
99. **if**((f==-1) && (r==-1))
100. {
101. printf("Deque is empty");
102. }
103. **else** **if**(f==r)
104. {
105. printf("\nThe deleted element is %d", deque[f]);
106. f=-1;
107. r=-1;
109. }
110. **else** **if**(f==(size-1))
111. {
112. printf("\nThe deleted element is %d", deque[f]);
113. f=0;
114. }
115. **else**
116. {
117. printf("\nThe deleted element is %d", deque[f]);
118. f=f+1;
119. }
120. }
122. // delete\_rear() function deletes the element from the rear
123. **void** delete\_rear()
124. {
125. **if**((f==-1) && (r==-1))
126. {
127. printf("Deque is empty");
128. }
129. **else** **if**(f==r)
130. {
131. printf("\nThe deleted element is %d", deque[r]);
132. f=-1;
133. r=-1;
135. }
136. **else** **if**(r==0)
137. {
138. printf("\nThe deleted element is %d", deque[r]);
139. r=size-1;
140. }
141. **else**
142. {
143. printf("\nThe deleted element is %d", deque[r]);
144. r=r-1;
145. }
146. }
148. **int** main()
149. {
150. insert\_front(20);
151. insert\_front(10);
152. insert\_rear(30);
153. insert\_rear(50);
154. insert\_rear(80);
155. display();  // Calling the display function to retrieve the values of deque
156. getfront();  // Retrieve the value at front-end
157. getrear();  // Retrieve the value at rear-end
158. delete\_front();
159. delete\_rear();
160. display(); // calling display function to retrieve values after deletion
161. **return** 0;
162. }

**Output:**



So, that's all about the article. Hope, the article will be helpful and informative to you.

# **What is a priority queue?**

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

### **Characteristics of a Priority queue**

A priority queue is an extension of a queue that contains the following characteristics:

* Every element in a priority queue has some priority associated with it.
* An element with the higher priority will be deleted before the deletion of the lesser priority.
* If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

**Let's understand the priority queue through an example.**

We have a priority queue that contains the following values:

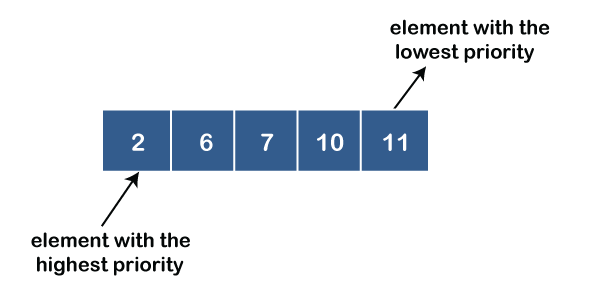
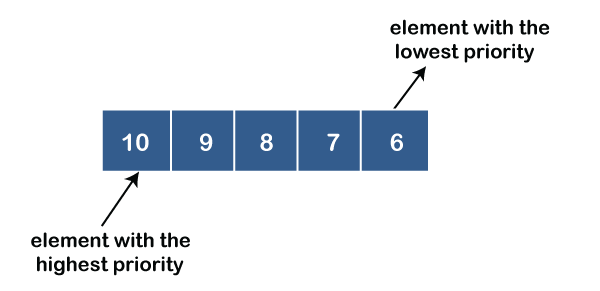
**1, 3, 4, 8, 14, 22**

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

* **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
* **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
* **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
* **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

### **Types of Priority Queue**

**There are two types of priority queue:**

* **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.  
  
* **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.  
  

### **Representation of priority queue**

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.



**Let's create the priority queue step by step.**

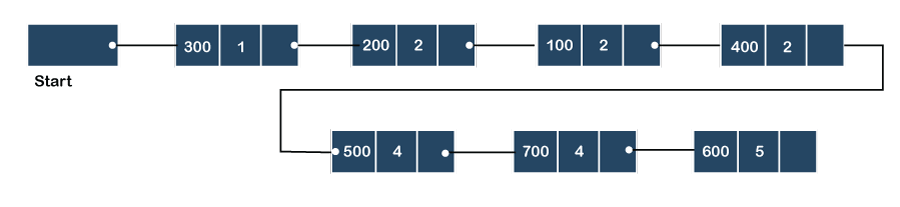
**In the case of priority queue, lower priority number is considered the higher priority, i.e.,** lower priority number = higher priority.

**Step 1:** In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

**Step 2:** After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

**Step 3:** After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

**Step 4:** After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



### **Implementation of Priority Queue**

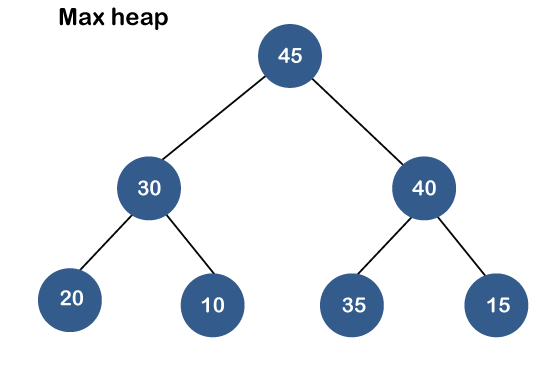
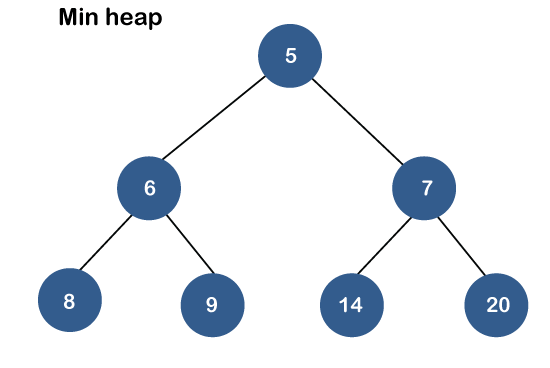
The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

**Analysis of complexities using different implementations**

|  |  |  |  |
| --- | --- | --- | --- |
| Implementation | add | Remove | peek |
| Linked list | O(1) | O(n) | O(n) |
| Binary heap | O(logn) | O(logn) | O(1) |
| Binary search tree | O(logn) | O(logn) | O(1) |

### **What is Heap?**

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

* **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.  
  
* **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.  
  

Both the heaps are the binary heap, as each has exactly two child nodes.

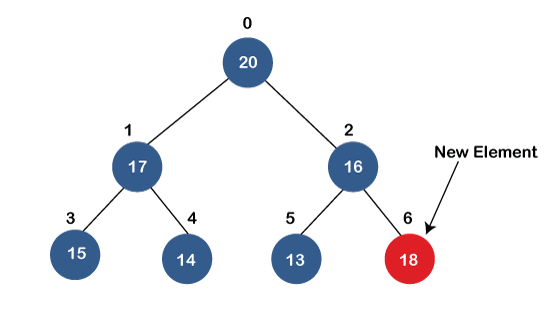
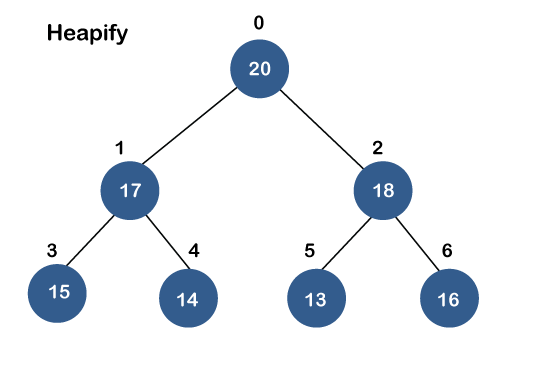
### **Priority Queue Operations**

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

* **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.

* **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

### **Applications of Priority queue**

**The following are the applications of the priority queue:**

* It is used in the Dijkstra's shortest path algorithm.
* It is used in prim's algorithm
* It is used in data compression techniques like Huffman code.
* It is used in heap sort.
* It is also used in operating system like priority scheduling, load balancing and interrupt handling.

**Program to create the priority queue using the binary max heap.**

1. #include <stdio.h>
2. #include <stdio.h>
3. **int** heap[40];
4. **int** size=-1;
6. // retrieving the parent node of the child node
7. **int** parent(**int** i)
8. {
10. **return** (i - 1) / 2;
11. }
13. // retrieving the left child of the parent node.
14. **int** left\_child(**int** i)
15. {
16. **return** i+1;
17. }
18. // retrieving the right child of the parent
19. **int** right\_child(**int** i)
20. {
21. **return** i+2;
22. }
23. // Returning the element having the highest priority
24. **int** get\_Max()
25. {
26. **return** heap[0];
27. }
28. //Returning the element having the minimum priority
29. **int** get\_Min()
30. {
31. **return** heap[size];
32. }
33. // function to move the node up the tree in order to restore the heap property.
34. **void** moveUp(**int** i)
35. {
36. **while** (i > 0)
37. {
38. // swapping parent node with a child node
39. **if**(heap[parent(i)] < heap[i]) {
41. **int** temp;
42. temp=heap[parent(i)];
43. heap[parent(i)]=heap[i];
44. heap[i]=temp;

47. }
48. // updating the value of i to i/2
49. i=i/2;
50. }
51. }
53. //function to move the node down the tree in order to restore the heap property.
54. **void** moveDown(**int** k)
55. {
56. **int** index = k;
58. // getting the location of the Left Child
59. **int** left = left\_child(k);
61. **if** (left <= size && heap[left] > heap[index]) {
62. index = left;
63. }
65. // getting the location of the Right Child
66. **int** right = right\_child(k);
68. **if** (right <= size && heap[right] > heap[index]) {
69. index = right;
70. }
72. // If k is not equal to index
73. **if** (k != index) {
74. **int** temp;
75. temp=heap[index];
76. heap[index]=heap[k];
77. heap[k]=temp;
78. moveDown(index);
79. }
80. }
82. // Removing the element of maximum priority
83. **void** removeMax()
84. {
85. **int** r= heap[0];
86. heap[0]=heap[size];
87. size=size-1;
88. moveDown(0);
89. }
90. //inserting the element in a priority queue
91. **void** insert(**int** p)
92. {
93. size = size + 1;
94. heap[size] = p;
96. // move Up to maintain heap property
97. moveUp(size);
98. }
100. //Removing the element from the priority queue at a given index i.
101. **void** **delete**(**int** i)
102. {
103. heap[i] = heap[0] + 1;
105. // move the node stored at ith location is shifted to the root node
106. moveUp(i);
108. // Removing the node having maximum priority
109. removeMax();
110. }
111. **int** main()
112. {
113. // Inserting the elements in a priority queue
115. insert(20);
116. insert(19);
117. insert(21);
118. insert(18);
119. insert(12);
120. insert(17);
121. insert(15);
122. insert(16);
123. insert(14);
124. **int** i=0;
126. printf("Elements in a priority queue are : ");
127. **for**(**int** i=0;i<=size;i++)
128. {
129. printf("%d ",heap[i]);
130. }
131. **delete**(2); // deleting the element whose index is 2.
132. printf("\nElements in a priority queue after deleting the element are : ");
133. **for**(**int** i=0;i<=size;i++)
134. {
135. printf("%d ",heap[i]);
136. }
137. **int** max=get\_Max();
138. printf("\nThe element which is having the highest priority is %d: ",max);

141. **int** min=get\_Min();
142. printf("\nThe element which is having the minimum priority is : %d",min);
143. **return** 0;
144. }

**In the above program, we have created the following functions:**

* **int parent(int i):** This function returns the index of the parent node of a child node, i.e., i.
* **int left\_child(int i):** This function returns the index of the left child of a given index, i.e., i.
* **int right\_child(int i):** This function returns the index of the right child of a given index, i.e., i.
* **void moveUp(int i):** This function will keep moving the node up the tree until the heap property is restored.
* **void moveDown(int i):** This function will keep moving the node down the tree until the heap property is restored.
* **void removeMax():** This function removes the element which is having the highest priority.
* **void insert(int p):** It inserts the element in a priority queue which is passed as an argument in a function**.**
* **void delete(int i):** It deletes the element from a priority queue at a given index.
* **int get\_Max():** It returns the element which is having the highest priority, and we know that in max heap, the root node contains the element which has the largest value, and highest priority.
* **int get\_Min():** It returns the element which is having the minimum priority, and we know that in max heap, the last node contains the element which has the smallest value, and lowest priority.

**Output**

